



Secure Boot from A to Z

Quentin Schulz
quentin@bootlin.com

Mylène Josserand
mylene@bootlin.com

© Copyright 2004-2018, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Speakers presentation

- ▶ Embedded Linux engineers at ~~Free Electrons~~ → Bootlin
 - ▶ Embedded Linux **expertise**
 - ▶ **Development**, consulting and training
 - ▶ Strong open-source focus
- ▶ Implemented full chain-of-trust on custom i.MX6 board
- ▶ Open-source contributors
- ▶ Living in **Toulouse**, south west of France



Disclaimer

- ▶ definitely **not** security experts
- ▶ presenting only one way to verify boot on a board based on a specific family of SoCs (though most parts can be applied to other boards)



Introduction



Who wants to verify the boot sequence and why?

- ▶ product vendors
 - ▶ make sure your devices are used the way they should be
 - ▶ not for a different purpose
 - ▶ not for running unapproved software (e.g. software limitations removed)
 - ▶ protect your consumers
- ▶ end users
 - ▶ make sure your system hasn't been tampered with
- ▶ basically, to make sure the binaries you're trying to load/boot/execute were built by a trustworthy person

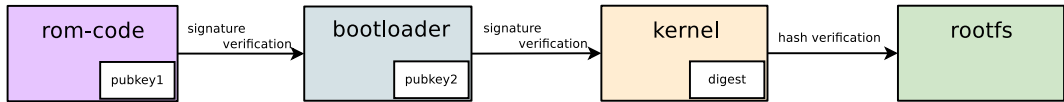


How does it work?

- ▶ everything is based on digital signature verification (\neq encryption)
- ▶ the first element in the boot process authenticates the second, the second the third, etc...
- ▶ called a chain-of-trust: if any element is authenticated but not sufficiently locked-down (e.g. console access in bootloader, root access in userspace), the device is not verified anymore



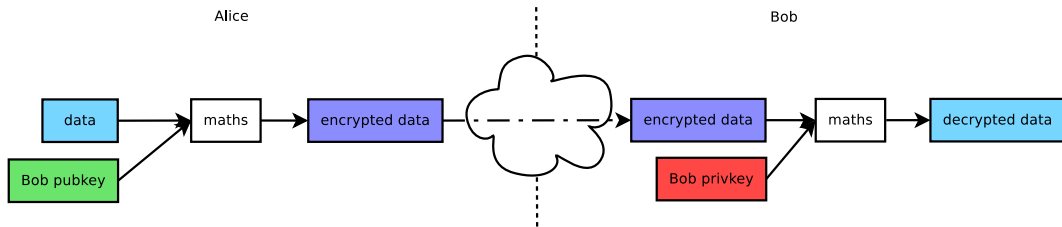
What does a chain-of-trust look like?



- ▶ every component is verified using its digital signature and a public key
- ▶ the rootfs integrity is verified using a hash mechanism
- ▶ our experience:
 - ▶ implemented chain-of-trust on custom i.MX6 boards
 - ▶ Quentin worked on the chain-of-trust from ROM code up to the kernel
 - ▶ Mylène worked on the root FS part of the chain-of-trust



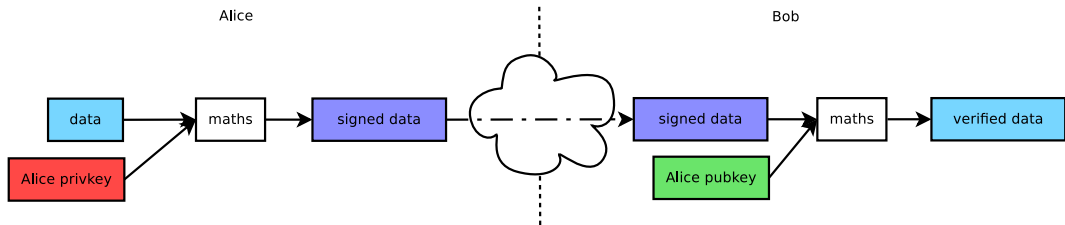
Mandatory Alice and Bob example: encryption



- ▶ provided Bob's public key is publicly available **anyone** (Alice, Charles, David, etc.) can send **encrypted data** to someone (Bob) that is the **only one** able to decrypt it



Mandatory Alice and Bob example: signature



- ▶ provided Alice's public key is publicly available, **anyone** (Bob, Charles, David, etc.) can verify that the **signed data** someone sent them is sent by the **only one** (Alice) able to sign it



Not inconsequential

- ▶ costly in terms of:
 - ▶ logistic and overall project complexity: whole architecture to create keys, build with the keys, ...
 - ▶ workflow complexity for developers: if the platform is locked down, need to re-sign the binary every time and validate the chain-of-trust
 - ▶ boot time (bunch of authentications to be made along the way to Linux prompt)
- ▶ you have to be extremely careful with your chain-of-trust and private keys so that none is broken or leaked



ROM code - Root of trust



Specific to the SoC



- ▶ need a way to store the public key(s) which will be used to decrypt the signature of the bootloader and make them tamper-proof
- ▶ each vendor can decide whatever medium they want to use to store the public keys
- ▶ microcode in charge of checking the signature is embedded in the ROM code
- ▶ different vendors: Xilinx, Tegra, Atmel, Freescale/NXP, Rockchip, ST, Samsung, ...



- ▶ the public key has to be stored on a non-volatile memory (NVM) accessible to the ROM code
- ▶ One-Time-Programmable (OTP) fuses are blown
- ▶ OTP fuses are silicon-expensive in terms of occupied area and store a relatively small amount of information
- ▶ a public key is at least 1 KiB
- ▶ less expensive to store only the hash of the public key in OTP, then compare it to the hash of the public key embedded in a given binary
- ▶ good idea to have multiple public keys so that if one private key is stolen/leaked/lost, we revoke it and we can use others and:
 1. not having a totally unverified device
 2. not having to brick the device



Secure boot sequence



ROM code:

- ▶ loads the bootloader in a secure space to avoid physical attacks
- ▶ loads the embedded public key
- ▶ checks the hash of the public key against the hash table in the OTP
- ▶ uses this verified public key to check the signature of the bootloader
- ▶ executes the bootloader binary
- ▶ called High Assurance Boot (HAB) for this SoC family



Preparing the board



- ▶ create the keys using NXP custom tool (Code Signing Tool)
- ▶ flash fuses from working unverified U-Boot using NXP-specific code and the fuse table returned by CST
- ▶ sign the bootloader using one of the keys whose hash is in the fuse table, using CST
- ▶ check status of bootloader `hab_status` which is NXP specific
- ▶ lock down bootloader loading by blowing the locking fuse



Bootloader authentication check



```
=> hab_status
Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66

----- HAB Event 1 -----
event data:
    0xdb 0x00 0x08 0x41 0x33 0x11 0xcf 0x00

STS = HAB_FAILURE (0x33)
RSN = HAB_INV_CSF (0x11)
CTX = HAB_CTX_CSF (0xCF)
ENG = HAB_ENG_ANY (0x00)

----- HAB Event 2 -----
event data:
[...]
```

```
=> hab_status
Secure boot disabled

HAB Configuration: 0xf0, HAB State: 0x66
No HAB Events Found!
```




Bootloader & kernel



Trust (almost) no one



- ▶ no point of having a secure bootloader if not authenticated by ROM code
- ▶ bootloader **has** to be sufficiently locked-down, otherwise there is no point authenticating it
- ▶ specific case of U-Boot mainline: **has** to be inaccessible by anyone (no console at all: `gd->flags |= GD_FLG_DISABLE_CONSOLE` in `board_early_init_f()`)
- ▶ under no circumstances should you trust anything that isn't in the U-Boot binary that is authenticated by the ROM code
- ▶ by default, the environment can be trusted **only** if it's in the U-Boot binary (`ENV_IS_NOWHERE`)
- ▶ pending patch in U-Boot to load only a handful of variables from another environment, limiting the attack vector
<https://patchwork.ozlabs.org/patch/855542/>



- ▶ U-Boot has DeviceTree Blob (DTB) support, used the same way the kernel does to probe drivers: according to the DT definition
- ▶ DTB can also be used to store a public key
- ▶ DTB is appended to the U-Boot binary and is thus affected by the computation of the hash used by the ROM code to authenticate the bootloader => can be trusted
- ▶ fitImage to have only one file containing binaries and signatures instead of lots of images to load
- ▶ `mkimage` (the tool to compile fitImages) has built-in support for signing of binaries hash



Key generation



- ▶ `openssl genrsa -out my_key.key 4096`
- ▶ `openssl req -batch -new -x509 -key my_key.key -out my_key.crt`
- ▶ `mkimage` requires certificate and private key files to be named the same



u-boot_pubkey.dts

```
/dts-v1/;
/ {
    model = "Keys";
    compatible = "vendor,board";
    signature {
        key-my_key {
            required = "image";
            algo = "sha1,rsa4096";
            key-name-hint = "my_key";
        };
    };
};
```

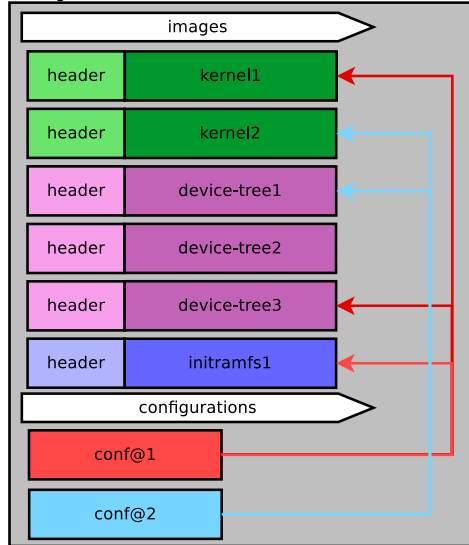
- ▶ `key-name-hint` and the suffix to the `key-` DT node has to be the same name as the one given to the key
- ▶ `required` is either `image` or `conf`, refer to `doc/uImage.FIT/signature.txt`



What's a fitImage?



fitImage



- ▶ several talks given to present the fitImage, the reasons behind and the challenges
- ▶ it's basically a container for multiple binaries with hashing and signature support
- ▶ it also supports forcing a few binaries to be loaded together,
- ▶ supports different architectures, OSes, image types, ... => can be found in `common/image.c`



```
/ {
  description = "fitImage for Foo revA and revB";
  #address-cells = <1>;
  images {
    kernel@1 {
      description = "Linux kernel";
      data = /incbin/("zImage");
      type = "kernel";
      arch = "arm";
      os = "linux";
      compression = "none";
      load = <0x10008000>;
      entry = <0x10008000>;
      signature@1 {
        algo = "sha1,rsa4096";
        key-name-hint = "my_key";
      };
    };
    fdt@1 {
      description = "DTB for Foo revA";
      data = /incbin/("foo-reva.dtb");
      type = "flat_dt";
      arch = "arm";
      compression = "none";
      signature@1 {
        algo = "sha1,rsa4096";
        key-name-hint = "my_key";
      };
    };
  };
  fdt@2 {
    description = "DTB for Foo revB";
    data = /incbin/("foo-revb.dtb");
    type = "flat_dt";
    arch = "arm";
    compression = "none";
    signature@1 {
      algo = "sha1,rsa4096";
      key-name-hint = "my_key";
    };
  };
  configurations {
    default = "conf@1";
    conf@1 {
      kernel = "kernel@1";
      fdt = "fdt@1";
    };
    conf@2 {
      kernel = "kernel@1";
      fdt = "fdt@2";
    };
  };
};
```



U-Boot & fitImage creation



```
#DTB compiled out-of-tree because we need to add the public key with \code{mkimage}
dtc u-boot_pubkey.dts -O dtb -o u-boot_pubkey.dtb
make CROSS_COMPILE=arm-linux-gnueabihf- foo_defconfig
make CROSS_COMPILE=arm-linux-gnueabihf- tools
tools/mkimage -f fitImage.its -K u-boot_pubkey.dtb -k /path/to/keys -r fitImage
make CROSS_COMPILE=arm-linux-gnueabihf- EXT_DTB=u-boot_pubkey.dtb
```




U-Boot required options



- ▶ `CONFIG_SECURE_BOOT=y` (specific to NXP)
- ▶ `#ifdef CONFIG_SECURE_BOOT`
`CSF CONFIG_CSF_SIZE`
`#endif`, at the beginning of the DCD file of your NXP board
- ▶ `CONFIG_OF_CONTROL=y`
- ▶ `CONFIG_DM=y, CONFIG_FIT=y, CONFIG_FIT_SIGNATURE=y`



fitImage booting



with a fitImage loaded @ 0x15000000:

```
=> bootm 0x15000000 #or bootm 0x15000000#conf@1 since conf@1 is the default
## Loading kernel from FIT Image at 15000000 ...
Using 'conf@1' configuration
Verifying Hash Integrity ... OK
Trying 'kernel@1' kernel subimage
  Description: Linux kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x150000e4
  Data Size: 7010496 Bytes = 6.7 MiB
  Architecture: ARM
  OS: Linux
  Load Address: 0x10008000
  Entry Point: 0x10008000
  Hash algo: sha1
  Hash value: 7d1fb52f2b8d1a98d555e01bc34d11550304fc26
  Sign algo: sha1,rsa4096:my_key
  Sign value: [redacted]
Verifying Hash Integrity ... sha1,rsa4096:my_key+ sha1+ OK
## Loading fdt from FIT Image at 15000000 ...
Using 'conf@1' configuration
Trying 'fdt@1' fdt subimage
[...]
Verifying Hash Integrity ... sha1,rsa4096:my_key+ sha1+ OK
Booting using the fdt blob at 0x156afd40
Loading Kernel Image ... OK
Loading Device Tree to 1fff2000, end 1ffff1ed ... OK

Starting kernel...
```



fitImage booting



with a fitImage loaded @ 0x15000000:

```
=> bootm 0x15000000 #or bootm 0x15000000#conf@1 since conf@1 is the default
## Loading kernel from FIT Image at 15000000 ...
Using 'conf@1' configuration
Verifying Hash Integrity ... OK
Trying 'kernel@1' kernel subimage
  Description: Linux kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x150000e4
  Data Size: 7010496 Bytes = 6.7 MiB
  Architecture: ARM
  OS: Linux
  Load Address: 0x10008000
  Entry Point: 0x10008000
  Hash algo: sha1
  Hash value: 7d1fb52f2b8d1a98d555e01bc34d11550304fc26
  Sign algo: sha1,rsa4096:my_key
  Sign value: [redacted]
Verifying Hash Integrity ... sha1,rsa4096:my_key+ sha1+ OK
## Loading fdt from FIT Image at 15000000 ...
Using 'conf@1' configuration
Trying 'fdt@1' fdt subimage
Verifying Hash Integrity ... sha1,rsa4096:my_key- Failed to verify required signature 'key-my_key'
error!
Unable to verify required signature for '' hash node in 'fdt@1' image node
Bad Data Hash
  Booting using the fdt blob at 0x156ba280
  Loading Kernel Image ... OK
ERROR: image is not a fdt - must RESET the board to recover.
FDT creation failed! hanging...### ERROR ### Please RESET the board ###
```



Root filesystem

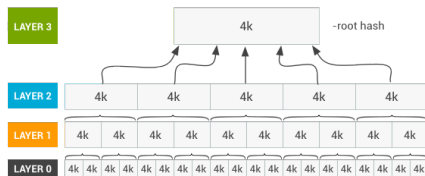


To have a verified root filesystem, we have chosen the following solutions:

- ▶ Have an unalterable filesystem:
 - ▶ read-only filesystem: impossible to modify it
=> squashfs: type for read-only filesystem
 - ▶ Not part of the secure-boot process but it was important for us
- ▶ Authenticate the rootfs
 - ▶ dm-verity:
 - ▶ infrastructure to check if the rootfs is the one we are expecting
=> authentication of the squashfs image
 - ▶ needs userspace applications to authenticate the system. Need to have these tools available
=> use an initramfs builtin as a first filesystem
 - ▶ the kernel is already in the chain of trust

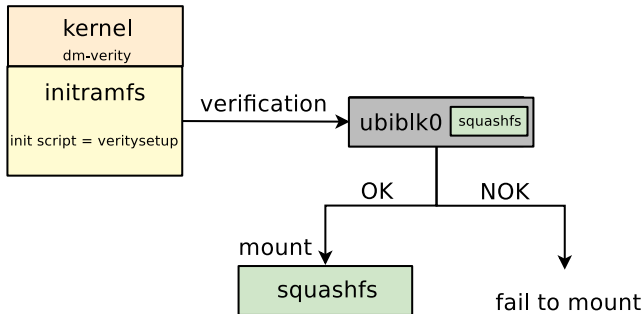


- ▶ Device-Mapper: infrastructure in the Linux kernel to create virtual layers of block devices
- ▶ Device-Mapper verity: provides integrity checking of block devices using kernel crypto API
- ▶ could hash the whole block device and compare it with the expected hash
- ▶ instead, use a cryptographic hash tree (Merkle tree)
- ▶ blocks are hashed and hash verified with hash tree **only on access**
- ▶ except the leaf nodes that are data, each node is the hash of its children. Until only one last hash => **root hash**
- ▶ needs userspace apps: **cryptsetup** provides different tools (*veritysetup*)





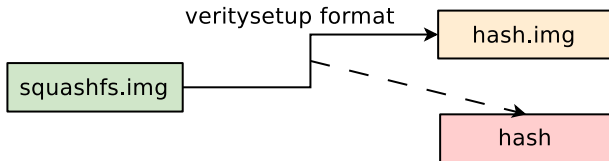
dm-verity in our case



- ▶ boot the kernel with initramfs
- ▶ have an init-script that uses `veritysetup` on block device (`ubiblk0`)
- ▶ `veritysetup`: a userspace application to authenticate devices according to `root_hash`
- ▶ if OK, verified squashfs available
- ▶ if NOK, fails to have squashfs
=> init stops here



dm-verity: create hash tree



command used:

veritysetup format

<data_device>

<hash_device>

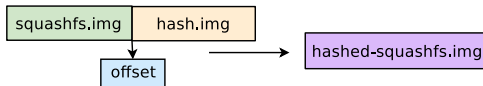
- ▶ veritysetup creates the hash tree (hash.img) and prints the root hash
- ▶ by default, the hash image is contained on another device/image than the one we want to authenticate



dm-verity: create hash tree



same device so concatenate the hash image in squashfs image



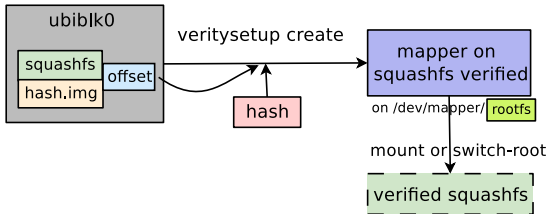
command used:

veritysetup format --hash-offset offset <data_device> <hash_device>

- ▶ not our use-case: want only one device
=> concatenate the hash image at the end of our squashfs image
- ▶ veritysetup has an option --hash-offset to locate the hash area in the same device/image



dm-verity: authenticate device



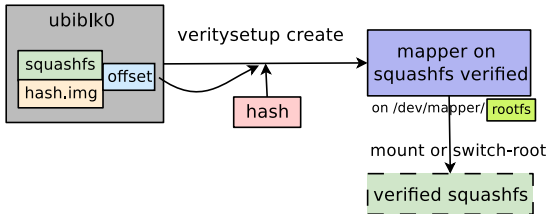
command used:

```
veritysetup create <name> --hash-offset <offset> <data_dev> <hash_dev> <hash>
```

- ▶ use `veritysetup` to authenticate the block device
- ▶ need the root hash and the offset (where to find the hash tree)
- ▶ if authentication is successful, can mount (or switch-root) the verified squashfs



dm-verity: authenticate device



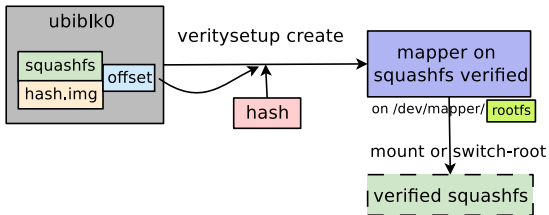
command used:

```
veritysetup create rootfs --hash-offset offset ubiblk0 ubiblk0 hash
```

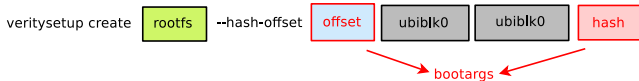
- ▶ use `veritysetup` to authenticate the block device
- ▶ need the root hash and the offset (where to find the hash tree)
- ▶ if authentication is successful, can mount (or switch-root) the verified squashfs



dm-verity: authenticate device



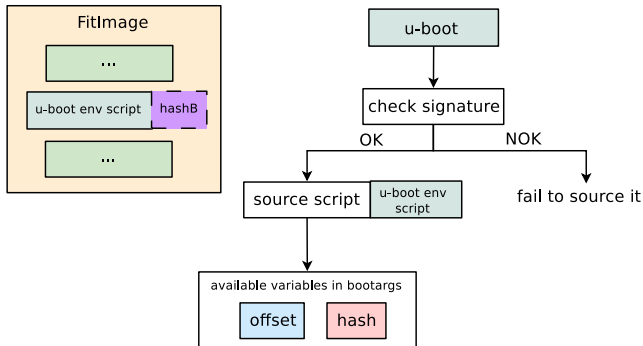
command used:



- ▶ use `veritysetup` to authenticate the block device
- ▶ need the root hash and the offset (where to find the hash tree)
- ▶ if authentication is successful, can mount (or switch-root) the verified squashfs



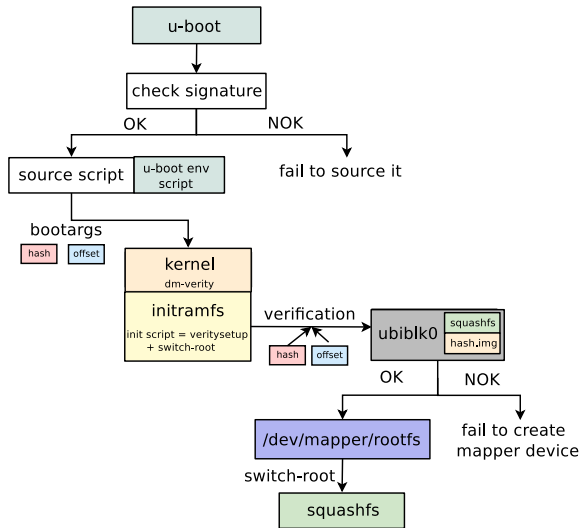
U-Boot: passing hash-offset



- ▶ create a U-Boot environment script
- ▶ but the U-Boot environment script can be attacked
- ▶ add this script in the FitImage
=> has a signature of the hash of the binary
- ▶ Once sourced, set bootargs to have offset and root hash



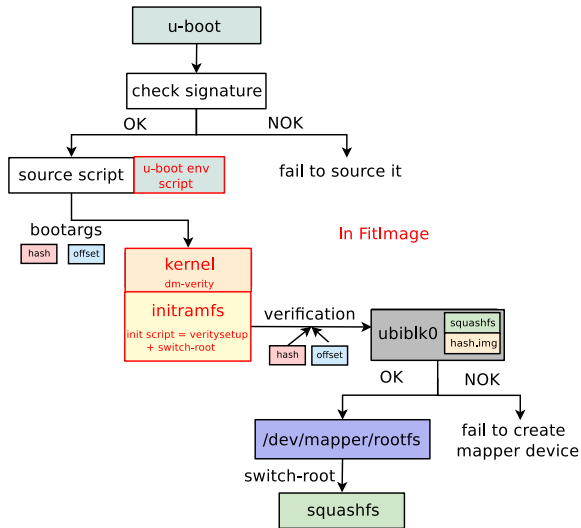
The final mechanism



- ▶ source U-Boot script to set bootargs with hash and offset
- ▶ bootargs read by Linux's init-script to retrieve hash/offset values
- ▶ used with veritysetup to authenticate the block device
- ▶ use switch-root tool to switch the rootfs from initramfs to squashfs



The final mechanism



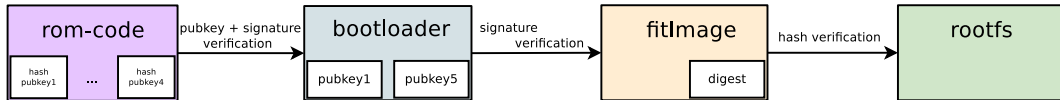
- ▶ source U-Boot script to set bootargs with hash and offset
- ▶ bootargs read by Linux's init-script to retrieve hash/offset values
- ▶ used with veritysetup to authenticate the block device
- ▶ use switch-root tool to switch the rootfs from initramfs to squashfs



Conclusion



Chain-of-trust completed





Painful integration into Yocto

- ▶ currently, to create a fitImage, the kernel recipe is required to inherit `kernel-fitimage` class
- ▶ it's done before the rootfs is created (because usually people want the kernel to be in `/boot`)
- ▶ U-Boot script needs to be in the fitImage
- ▶ U-Boot script has to be created after the squashfs rootfs to retrieve the root hash
- ▶ and that's how you end up with a dependency loop in Yocto :)
- ▶ wrote a new image and class to work around this issue



Read-Write filesystems

- ▶ our use case was very specific: read-only root filesystem, but one might want a read-write filesystem
- ▶ if not critical (depends on your use case, e.g. logs, user data, etc...), mount it along side your read-only authenticated rootfs
- ▶ if critical, have a look at IMA/EVM
 - ▶ <http://kernsec.org/files/lss2015/ima-applications-slides.pdf>
 - ▶ <https://lwn.net/Articles/488906/>



Remember about trusting no-one?

- ▶ secure boot vulnerabilities in ROM code of i.MX6, i.MX50, i.MX53, i.MX7, i.MX28 and Vybrid families publicly disclosed July 17th, 2017
 - ▶ <https://community.nxp.com/docs/DOC-334996>
- ▶ Know your threat model, nothing is 100% secure,
 - ▶ Tutorial: Introduction to Reverse Engineering by Mike Anderson

Questions?

Quentin Schulz

quentin@bootlin.com

Mylène Josserand

mylene@bootlin.com

Slides under CC-BY-SA 3.0

<https://bootlin.com/pub/conferences/2018/elc/josserand-schulz-secure-boot>

Support our crowdfunding campaign to develop
an upstream Linux kernel driver for Allwinner VPU

<https://bootlin.com/blog/allwinner-vpu-crowdfunding/>

